A Local Graph-rewriting System for Deciding Equality in Sum-product Theories (Extended Abstract)

José Bacelar Almeida, Jorge Sousa Pinto, and Miguel Vilaça [1,2,3]

Departamento de Informática
Universidade do Minho
4710-057 Braga, Portugal

1 Introduction

The *point-free* style of programming [1] has been defended as a good choice for reasoning about functional programs. However, when one actually tries to construct a decision procedure for the associated equational theory, one faces problems, even when small fragments of the theory are considered.

In this paper we outline how a graph-based decision procedure can be given for the functional calculus with sums and products (but no exponentials – the expressions we use here can not really be seen as a programming language). We show in turn how the system covers *reflexivity* equational laws, *fusion* laws, and *cancelation* laws.

The decision procedure has interest independently of our initial motivation. The term language (and its theory) can be seen as the internal language of a category with binary products and coproducts. A standard approach based on term rewriting would work *modulo* a set of equations; the present work proposes a simpler approach, based on graph-rewriting.

2 The Term Language and Theory

Consider the following language $\mathcal{T}_{\mathrm{PF}}$ for types and terms:

$$\begin{aligned}\mathsf{Type} ::= {} & A \mid \mathsf{Type} \times \mathsf{Type} \mid \mathsf{Type} + \mathsf{Type} \\ \mathsf{Term} ::= {} & C^{\mathsf{Type},\mathsf{Type}} \mid \mathsf{id}^{\mathsf{Type}} \mid \mathsf{Term} \cdot \mathsf{Term} \mid \langle \mathsf{Term}, \mathsf{Term} \rangle \mid {\pi_1}^{\mathsf{Type},\mathsf{Type}} \mid \\ & {\pi_2}^{\mathsf{Type},\mathsf{Type}} \mid [\mathsf{Term}, \mathsf{Term}] \mid {\mathrm{i}_1}^{\mathsf{Type},\mathsf{Type}} \mid {\mathrm{i}_2}^{\mathsf{Type},\mathsf{Type}}\end{aligned}$$

[1] Email:jba@di.uminho.pt
[2] Email:jsp@di.uminho.pt
[3] Email:jmvilaca@di.uminho.pt

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

where A is a set of *base types* and $C^{\mathsf{Type},\mathsf{Type}}$ is a set of *constant functions* (we assume that the sets in this indexed family are pairwise disjunct – thus a constant symbol uniquely determines its indexing types).

To each term we associate a *domain* and a *codomain* type – we denote $f : A \to B$ the assertion that term f has domain A and codomain B. The typing rules associated to the language are the following

$$\frac{}{c^{A,B} : A \to B}\ c^{A,B} \in C^{A,B} \qquad \frac{}{\mathsf{id}^A : A \to A} \qquad \frac{f : A \to B \qquad g : B \to C}{g \cdot f : A \to C}$$

$$\frac{f : A \to C \qquad g : B \to C}{[f,g] : (A + B) \to C} \qquad \frac{}{{\pi_1}^{A,B} : (A \times B) \to A} \qquad \frac{}{{\pi_2}^{A,B} : (A \times B) \to B}$$

$$\frac{f : A \to B \qquad g : A \to C}{\langle f,g\rangle : A \to (B \times C)} \qquad \frac{}{{\mathsf{i}_1}^{A,B} : A \to (A + B)} \qquad \frac{}{{\mathsf{i}_2}^{A,B} : B \to (A + B)}$$

In the following, when referring to a term we assume its well-typedness. We will ommit the type superscripts, which can be inferred from the context.

The type constructors $\times$ and $+$ are characterized through their universal properties. These, in turn, may be captured by the following set of equations:

Composition	$\mathsf{id} \cdot f = f \cdot \mathsf{id} = f$	$(f \cdot g) \cdot h = f \cdot (g \cdot h)$
Reflexivity laws	$\langle \pi_1, \pi_2 \rangle = \mathsf{id}$	$[\mathsf{i}_1, \mathsf{i}_2] = \mathsf{id}$
Fusion laws	$\langle f,g\rangle \cdot h = \langle f \cdot h, g \cdot h\rangle$	$f \cdot [g,h] = [f \cdot g, f \cdot h]$
Cancelation laws	$\pi_1 \cdot \langle f,g\rangle = f$	$[f,g] \cdot \mathsf{i}_1 = f$
	$\pi_2 \cdot \langle f,g\rangle = g$	$[f,g] \cdot \mathsf{i}_2 = g$

Deciding equality under the theory defined by these equations requires producing a decision procedure. The simplest way to accomplish this is to orient the equations to obtain a confluent, terminating rewriting system (possibly by means of a completion process). Unfortunately, in this case it is not possible to conduct this program. Even considering the multiplicative fragment alone (i.e. ignoring the terms that involve sums), we face problems when constructing a rewriting system from the corresponding laws.

3 Difficulties

In the multiplicative sub-system, the orientation *left to right* seems sensible, but creates unsolvable critical pairs induced by the reflection laws. To illustrate this problem consider the derived law (*surjective pairing*) $f = \mathsf{id} \cdot f = \langle \pi_1, \pi_2 \rangle \cdot f = \langle \pi_1 \cdot f, \pi_2 \cdot f \rangle$. Both extremes of the equality chain are in normal form with respect to the rewrite system obtained, thus it fails to be complete.

A closer look at the reflection law gives us a hint of what the problem is – it drops from the term structural information that is essential for the

confluence of the system. An approach to overcoming this problem consists in imposing that all the rewrites preserve the structural information (allowing for the reconstruction of types), together with the proviso that the starting term contains all the structural information to reconstruct its type structure. In practice, we can drop *identities* from the language, except at base types, and the reflexivity law can be dropped from the rewriting system – it becomes a rule for defining identities of structured types. As an example, the identity of type $(A \times B) \times C$ is defined as $\langle\langle \pi_1, \pi_2 \rangle \,.\, \pi_1, \pi_2 \rangle$.

Constant functions should also carry their structural information. To avoid restricting constant functions to base types, we may instead exhibit that information by composing the functions with appropriate identities (defined as above). This means that a normal form of a constant function f with codomain $A \times B$ is the normal form of $\langle \pi_1, \pi_2 \rangle \cdot f$, that is $\langle \pi_1 \cdot f, \pi_2 \cdot f \rangle$. Equations like surjective pairing are then satisfied by construction.

Restricting our attention to the additive fragment will lead to dual arguments. However, when both products and sums are considered, a simple rewriting approach faces irremediable problems: not only does associativity of composition become a concern (there no longer exists a sensible orientation for it), but products and sums interact in such a symmetrical way that the rewriting system cannot "choose" a certain form to the detriment of its dual.

To see an example that illustrates this last observation, consider the equality derivation (known as the *exchange law*), $\langle [f, g], [h, k] \rangle = \langle [f, g], [h, k] \rangle \cdot [\mathrm{i}_1, \mathrm{i}_2] = [\langle [f, g], [h, k] \rangle \cdot \mathrm{i}_1, \langle [f, g], [h, k] \rangle \cdot \mathrm{i}_2] = [\langle [f, g] \cdot \mathrm{i}_1, [h, k] \cdot \mathrm{i}_1 \rangle, \langle [f, g] \cdot \mathrm{i}_2, [h, k] \cdot \mathrm{i}_2 \rangle] = [\langle f, h \rangle, \langle g, k \rangle]$. To decide equality of the sum-product theory through a rewriting system, one must work modulo an appropriate equational theory that handles these equalities (see for instance [3]).

In this paper we follow a totally different approach: the graph-rewriting system introduced in the next sections captures associativity of composition for free, and treats the interaction between the multiplicative and the additive fragments adequately (for instance the two sides of the exchange law have the same normal form). Reflexivity is treated as outlined above.

4 Sum-product Nets

Sum-product Nets will be built from instances of *symbols*; each symbol has an associated number of input ports (or *arity*) and number of output ports (or *co-arity*). We organize these symbols in *dual* pairs where the arity and co-arity are exchanged. These symbols are:

- a *duplicator* symbol with arity 1 and co-arity 2, depicted $\wedge$; its dual is the *co-duplicator*, depicted $\vee$;
- a *makepair* symbol with arity 2 and co-arity 1, depicted $(,)$; its dual is *choice* and depicted ?;
- two *pair projection* symbols with arity 1 and co-arity 1, depicted π_1 and π_2;

their duals are the *choice injections* depicted i_1 and i_2;

- an *eraser* symbol with arity 1 and co-arity 0, depicted ε; the dual *co-eraser* is depicted $\ni$.
- a *cancel* symbol with arity and co-arity 1, depicted ■; its dual *co-cancel* is depicted □.

A *Net* is a tuple (S, E, I, O) where S is a set of occurrences of symbols, E is a set of *edges*, and I, O are two sets of *input ports* and *output ports* of the net. Input and output ports of the net do not belong to any symbol occurrence. Let S^I, S^O denote respectively the sets of input and output ports of the symbol occurrences in S. Then each edge in E connects a port in $S^O \cup I$ (the output port of some symbol occurrence or an input of the net) to a port in $S^I \cup O$ (the input port of some symbol occurrence or an output of the net). Every port in $S^I \cup S^O \cup I \cup O$ belongs to exactly one edge. In the rest of the paper we refer to occurrences of symbols as *nodes*.

In what follows, $\wedge$, $\vee$, □ and *cocancel* nodes in a net will be labelled with indexes. Indexes are pairs of bit strings for $\wedge$ and $\vee$, and bit strings for □ and ■. These will be used to control the duplication and mutual annihilation of nodes in the reduction system presented in section 5.

A net is *well-typed* is there exists a labelling of the input and output ports of each of its nodes with a type, such that every edge connects equally labelled ports, and the constraints shown in Figure 1 hold for every node.

A *position* is a pair of bit strings (α, β), depicted as $\alpha \cdot \beta$. A net is *well-formed* if there exists a labelling of the input and output ports of each of its nodes with a position, such that every edge connects equally labelled ports, and the constraints also shown in Figure 1 hold for every node. Well-formedness imposes a structural invariant on nets.

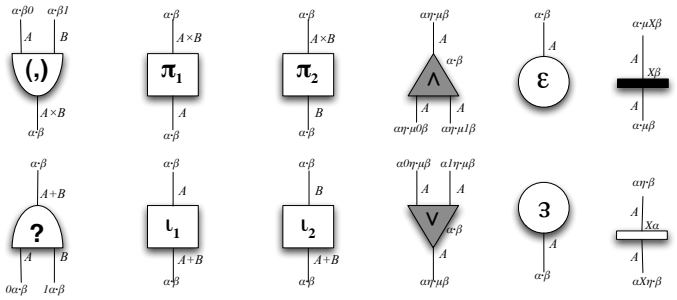


Fig. 1. Typing and Positioning Constraints

Definition 4.1 A *sum-product net* is an acyclic, well-typed and well-formed net with a single input and output, both labelled with empty positions.

Figure 2 contains examples of nets that are not sum-product nets: the first net is not well-typed; the second is not well-formed; the third net has a cycle.

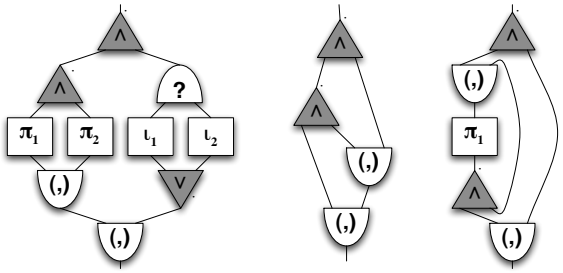


Fig. 2. Examples of Nets

5 Sum-product Net Rewriting

A local graph-rewriting system will now be given for sum-product nets. We first need to establish an appropriate notion of graph-rewriting rule: both the left-hand side (LHS) and the right-hand side (RHS) of the rule are finite nets, such that the sets of input and output ports are the same in both nets (in other words the rule preserves the interface of the net). Moreover both the LHS and RHS nets are well-typed and well-formed, the rule preserves type and position labellings of the inputs and outputs, and does not introduce cycles.

The application of a rule in a typed net replaces any subnet matching its LHS by its RHS; the conditions above guarantee that there will be no edges left dangling. The system introduced below enjoys additionally the following:

- There are no two rules in the system with the same LHS, or such that the LHS of a rule is a subnet of the LHS of the other;
- The RHS of each rule does not contain as a subnet the LHS of another rule;
- The set of rules is *dual-complete*: the dual of each rule is also in the system.

This has some of the defining properties of an interaction net system [2]; further requirements of such a system are that each node should have a distinguished principal port, and the LHS of every rule should consist of two nodes with an edge connecting both principal ports. This requirement is sufficient to guarantee strong local confluence, which is not a property of our system.

The rules are introduced in two sets: a first set allows to decide the theory minus the cancelation laws; a second set of rules addresses these laws.

Fusion Rules.

Fusion is accomplished by the interaction with (co-)duplicators. Intuitively, a duplicator interacting with a net should perform a copy of that net. However, this "duplication" should take in account that we intend it to be performed locally, i.e. the (co-)duplicators interact only with individual agents. Moreover, both kinds of fusion (additive and multiplicative) can occur simultaneously and thus some care must be taken in order to avoid interferences in the process. Figure 3 shows the rules for this fragment of the system (we omit those rules that can be obtained by duality).

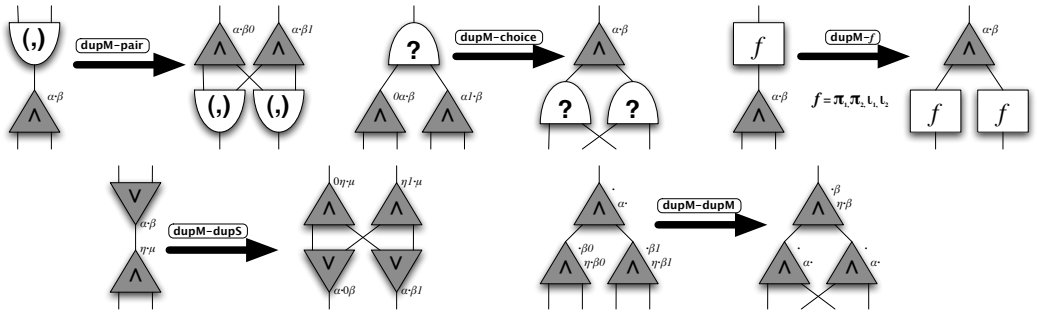
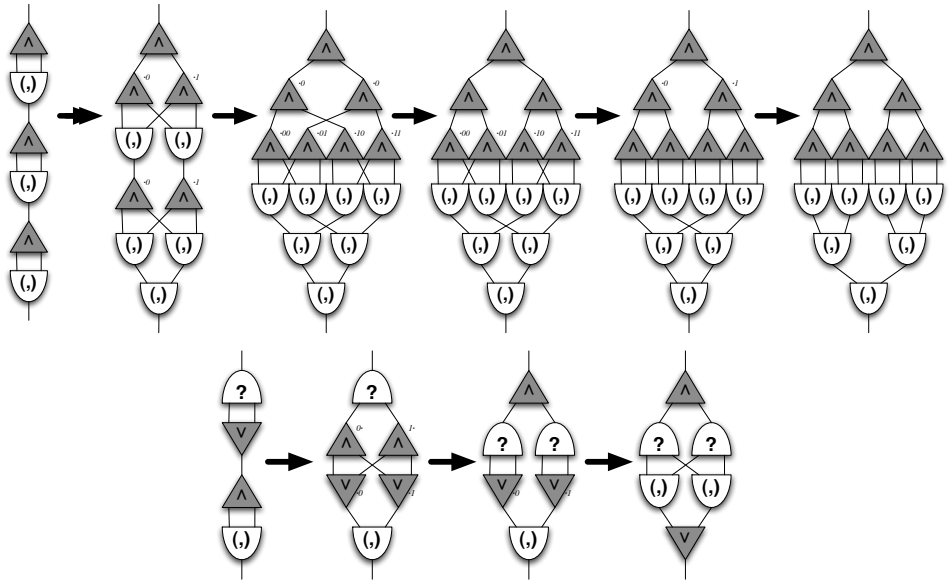


Fig. 3. Fusion Rules

Two example reductions are shown below. The fundamental role taken by indexes in controlling commutations is evident (rules `dupM-dupS` and `dupM-dupM`).



Cancelation Rules.

Cancelation rules are essentially not local – taking the multiplicative rule, apart from the removal of the pair constructor, it should also remove the top duplicator and discharge a whole sub-net. On the other hand, it should not interfere with fusion indexes for duplicators that might be in transit through the net. These considerations lead to the introduction of erasing ($\varepsilon/\ni$) and control nodes (*cancel* and *co-cancel*).

The set of rules for cancelation is given in Figure 4. Most rules concern the movement induced on *cancel* and *cocancel* nodes to promote their annihilation. The role of *cancel*/*cocancel* nodes is to correct the indexes of $\wedge$ and $\vee$ nodes (resulting from previous fusions) that cross the cancelation points. The use of X in the indexes denotes an arbitrary bit. Indexes are adjusted by either discarding an appropriate bit or leaving them unchanged – the choice is made based on the size of indexes. Observe that this set is complemented with garbage-collection rules for ε and $\ni$ nodes (not shown), which only remove

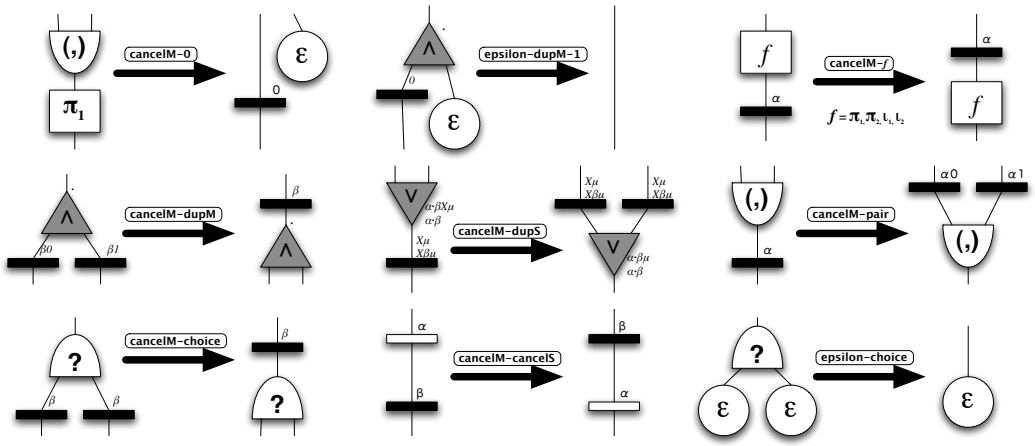


Fig. 4. Cancelation Rules

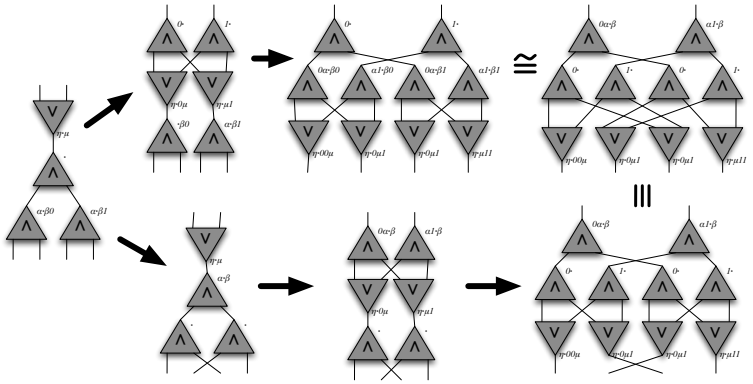


Fig. 5. Example critical pair for the rules in Figure 3

$\wedge/\vee$ nodes with empty indexes. This guarantees that the nodes "in transit" perform the necessary commutations. The annihilation of cancel/cocancel agents requires the introduction of indexes for these agents (details will be given in the full version of the paper [4]).

Properties.

It is straightforward to see that the system is *strongly normalizing* (in general $\wedge$ and $\square$ nodes go up; $\vee$ and $\blacksquare$ nodes go down; commutations between three $\wedge$ or $\vee$ nodes impose unique configurations). The system is also *confluent*: only one critical pair requires specific structural arguments; all the remaining ones are resolved locally modulo a notion of equivalence ($\cong$) that allows for a simple and general proof, even if relying on the structural invariant of sum-product nets. Figure 5 shows an example critical pair.

This reduction system thus induces the following definition of equivalence of sum-product nets. Let $\equiv$ denote structural equality of nets.

Definition 5.1 Two sum-product nets G_1, G_2 are *equivalent*, written $G_1 = G_2$, if there exist G_1', G_2' such that $G_1 \longrightarrow^* G_1'$ and $G_2 \longrightarrow^* G_2'$, and $G_1' \equiv G_2'$.

6 Term Nets

We now give a type-directed translation $\mathbf{T}(\cdot)$ from terms of $\mathcal{T}_{\mathrm{PF}}$ into sum-product nets. When a smaller net is used to construct some other net, we assume that the input and output in the initial net are removed. We also assume that a new pair of input/output ports and corresponding edges are introduced in the new net. The indexes of new nodes are initially empty.

Identity

- $\mathbf{T}(\mathsf{id} : A \to A)$, where A is a base type, is defined as the sum-product net consisting of a single edge connecting the input to the output;
- $\mathbf{T}(\mathsf{id} : A \times B \to A \times B)$ is the sum-product net I obtained by introducing 4 new nodes, $\wedge$, π_1, π_2, and $(,)$, and new edges connecting the first (resp. second) output of $\wedge$ to the input of π_1 (resp. π_2), the output of π_1 (resp. π_2) to the input of I_A (resp. I_B), and the output of I_A (resp. I_B) to the first (resp. second) input of $(,)$, where $I_A = \mathbf{T}(\mathsf{id} : A \to A)$ and $I_B = \mathbf{T}(\mathsf{id} : B \to B)$; and finally setting the input of I to be the input of $\wedge$ and the output of I to be the output of $(,)$.
- $\mathbf{T}(\mathsf{id} : A + B \to A + B)$ is the sum-product net I obtained by introducing 4 new nodes, ?, i_1, i_2, and $\vee$, and new edges connecting the first (resp. second) output of ? to the input of I_A (resp. I_B), the output of I_A (resp. I_B) to the input of i_1 (resp. i_2), and the output of i_1 (resp. i_2) to the first (resp. second) input of $\vee$, where $I_A = \mathbf{T}(\mathsf{id} : A \to A)$ and $I_B = \mathbf{T}(\mathsf{id} : B \to B)$; and finally setting the input of I to be the input of ? and the output of I to be the output of $\vee$.

Composition

- $\mathbf{T}(u\,.\,t : A \to C)$ is the sum-product net V obtained by connecting an edge from the output of T to the input of U, where $T = \mathbf{T}(t : A \to B)$ and $U = \mathbf{T}(u : B \to C)$. Naturally, the input of T becomes the input of V, and the output of U becomes the output of V.

Constant Function

- $\mathbf{T}(\pi_1 : A \times B \to A)$ is the net P_1 obtained by introducing a new node π_1 and a new edge connecting its output to the input of I_A, where $I_A = \mathbf{T}(\mathsf{id} : A \to A)$, and setting the input of P_1 to be the input of π_1 and the output of P_1 to be the output of I_A.
- $\mathbf{T}(\pi_2 : A \times B \to B)$ is the net P_2 obtained by introducing a new node π_2 and a new edge connecting its output to the input of I_B, where $I_B = \mathbf{T}(\mathsf{id} : B \to B)$, and setting the input of P_2 to be the input of π_2 and the output of P_2 to be the output of I_B.
- $\mathbf{T}(\mathrm{i}_1 : A \to A + B)$ is the net I_1 obtained by introducing a new node i_1 and a new edge connecting the output of I_A to the input of i_1, where $I_A = \mathbf{T}(\mathsf{id} : A \to A)$, and setting the input of I_1 to be the input of I_A and the output of I_1 to be the output of i_1.
- $\mathbf{T}(\mathrm{i}_1 : B \to A + B)$ is the net I_2 obtained by introducing a new node i_2 and a new edge connecting the output of I_B to the input of i_2, where

$I_B = \mathbf{T}(\mathsf{id} : B \to B)$, and setting the input of I_2 to be the input of I_B and the output of I_2 to be the output of i_2.

Split

Let G be the sum-product net obtained by introducing two new $\wedge$ and $(,)$ nodes, and 4 new edges connecting the outputs of $\wedge$ to the inputs of T and U, and the outputs of T and U to the inputs of $(,)$, where $T = \mathbf{T}(t : E \to A)$ and $U = \mathbf{T}(u : E \to B)$; the input of $\wedge$ becomes the input of G and the output of $(,)$ becomes the output of G. Then:

- $\mathbf{T}(\langle t, u\rangle : E \to A \times B)$, with $E = C + D$, is the sum-product net G' obtained by constructing the net $I = \mathbf{T}(\mathsf{id} : C + D \to C + D)$, and an edge connecting its output to the input of G, setting the input of G' to be the input of I, and the output of G' to be the output of G.
- $\mathbf{T}(\langle t, u\rangle : E \to A \times B)$, where E is not of the form $C + D$, is just G.

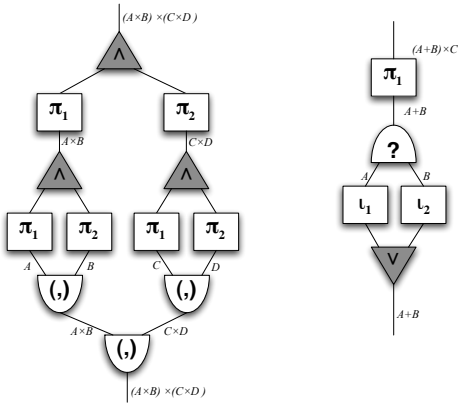
Either

Let G be the sum-product net obtained by introducing two new ? and $\vee$ nodes, and 4 new edges connecting the outputs of ? to the inputs of T and U, and the outputs of T and U to the inputs of $\vee$, where $T = \mathbf{T}(t : A \to E)$ and $U = \mathbf{T}(u : B \to E)$; then the input of ? becomes the input of G and the output of $\vee$ becomes the output of G. We have: cancela

- $\mathbf{T}([t, u] : A + B \to E)$, where $E = C \times D$, is the sum-product net G' obtained by constructing the net $I = \mathbf{T}(\mathsf{id} : C \times D \to C \times D)$, and an edge connecting the output of G to the input of I; the input of G' is the input of G, and the output of G' is the output of I.
- $\mathbf{T}([t, u] : A + B \to E)$, where E is not of the form $C \times D$, is just G.

Definition 6.1 The class of sum-product nets constructed by the translation $\mathbf{T}(\cdot)$ are designated *term nets.*

The term nets $\mathbf{T}(\mathsf{id} : (A \times B) \times (C \times D) \to (A \times B) \times (C \times D))$ and $\mathbf{T}(\pi_1 : (A + B) \times C \to A + B)$ are shown below as examples.



It is straightforward to see that $\mathbf{T}(t : A \to B)$ is indeed a term net with input of type A and output of type B. A distinctive feature of the translation is that two differently-typed, syntactically equal terms may be translated as different term nets. The translation introduces in the nets sufficient structural

information to allow for the typing information to be discarded. The principal type of the term represented by a net can always be uniquely determined.

Deciding Equality.

We may now establish the main result relating the equational theory and the graphical system.

Proposition 6.2 (Soundness and Completeness) *Let t, u be $\mathcal{T}_{\mathrm{PF}}$ terms. Then $t = u \iff \mathbf{T}(t) = \mathbf{T}(u)$.*

The *soundness* part can be proved by induction on the definition of equality in $\mathcal{T}_{\mathrm{PF}}$. The completeness part is proved using a path semantics for graphs.

7 Conclusions and Further Work

The main features of the translation $\mathbf{T}(\cdot)$ and the graph-rewriting system are:

- The translation directly captures the reflexivity laws, because it expands identities according to their types.
- To ensure that the fusion laws are effectively captured, commutations between configurations involving 3 nodes (rules dupM-dupM, dupM-choice, dupS-dupS and pair-dupS) are allowed, regulated by an indexing scheme.
- Finally, this indexing scheme is capable of handling fusions in terms such as $\langle a, b\rangle\,.[c, d]$, which may happen in two directions. Such a fusion results in a net which is no longer a term net.

An adequate treatment of the exponential fragment of the calculus is the next obvious step. This introduces new problems, related to the work on encodings of the λ-calculus into interaction nets. The initial and terminal objects and their associated morphisms can easily be incorporated in our system.

We also intend to use this graph-rewriting system in the context of a visual language for functional programming.

References

[1] Bird, R., de Moor, O.: *Algebra of Programming*, Prentice Hall, 1997.

[2] Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.

[3] J. R. B. Cockett and R. A. G. Seely Finite sum - product logic. In *Theory and Applications of Categories*, Vol. 8, 2001, No. 5, pp 63-99.

[4] J. B. Almeida, J. S. Pinto and J. M. Vilaça. A Local Graph-rewriting System for Deciding Equality in Sum-product Theories. *DI-PURe Technical Report, Universidade do Minho*. Available from `http://wiki.di.uminho.pt/bin/view/PURe/Publications`.